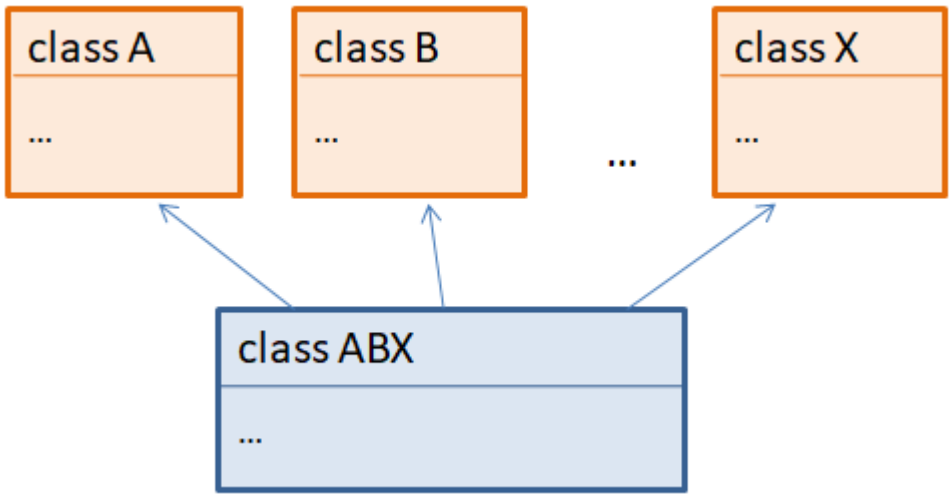
До сих пор мы с вами наследовали дочерний класс от одного базового. И это относительно частая ситуация. Однако иногда требуется выполнять наследование сразу от нескольких базовых классов. Такой подход называется **множественным наследованием**.



Надо сказать, что не все языки программирования предоставляют такой функционал. **Например, в Java множественное наследование запрещено**. Правда, там есть классы специальных видов – интерфейсы, которые несколько сглаживают это ограничение. С чем это связано? **При наличии нескольких базовых классов можно легко нарушить логику работы всей этой конструкции, так как классы не редко разрабатываются разными программистами и учесть их согласованную работу становится не так то просто**. Или же, один и тот же программист спустя определенное время может легко внести в один из классов такие правки, которые приведут к краху всей программы. Поэтому применять множественное наследование следует с крайней осторожностью и лучше здесь придерживаться устоявшихся подходов к построению таких конструкций.

Тем не менее, язык C++, наряду с другими ЯП, позволяет нам наследоваться сразу от нескольких классов. И в ряде случаев это бывает весьма полезной возможностью. Поэтому давайте внимательно рассмотрим этот механизм.

На предыдущих занятиях у нас с вами использовался базовый класс GeomBase и дочерний Line. Запишем их в следующем виде:

```cpp
class GeomBase {
protected:
    int x0{0}, y0{0}, x1{0}, y1{0};
public:
```

```
    GeomBase(int a = 0, int b = 0, int c = 0, int d = 0)
        : x0(a), y0(b), x1(c), y1(d)
        { }

    void set_coords(int x0, int y0, int x1, int y1)
    {
        this->x0 = x0; this->y0 = y0;
        this->x1 = x1; this->y1 = y1;
    }

    virtual void draw() const = 0;
};

class Line : public GeomBase {
private:
    double length{0.0};
public:
    Line(int a = 0, int b = 0, int c = 0, int d = 0) : GeomBase(a, b, c, d)
        { }

    virtual void draw() const
        { printf("Line: %d, %d, %d, %d\n", x0, y0, x1, y1); }
};
```

Здесь все должно быть вам знакомо и понятно. Но допустим, что затем, мы решили разработать функционал для сохранения геометрических фигур в файл. Так как это самостоятельная задача, то ее целесообразно выделить в отдельный класс. Для простоты опишем его следующим образом (в самом начале):

```language-cpp
#include <fstream>

class GeomSerialize {
protected:
    bool fl_saved {false};
public:
    virtual void save(std::ostream& os) const = 0;
    virtual void load(std::istream& is) = 0;
};
```

И, затем, унаследует класс Line сразу от двух классов:

```language-cpp
class Line : public GeomBase, public GeomSerialize {
...
};
```

Обратите внимание, как прописано множественное наследование. Перед каждым классом указывается его режим наследования (public), а сами классы записаны через запятую.

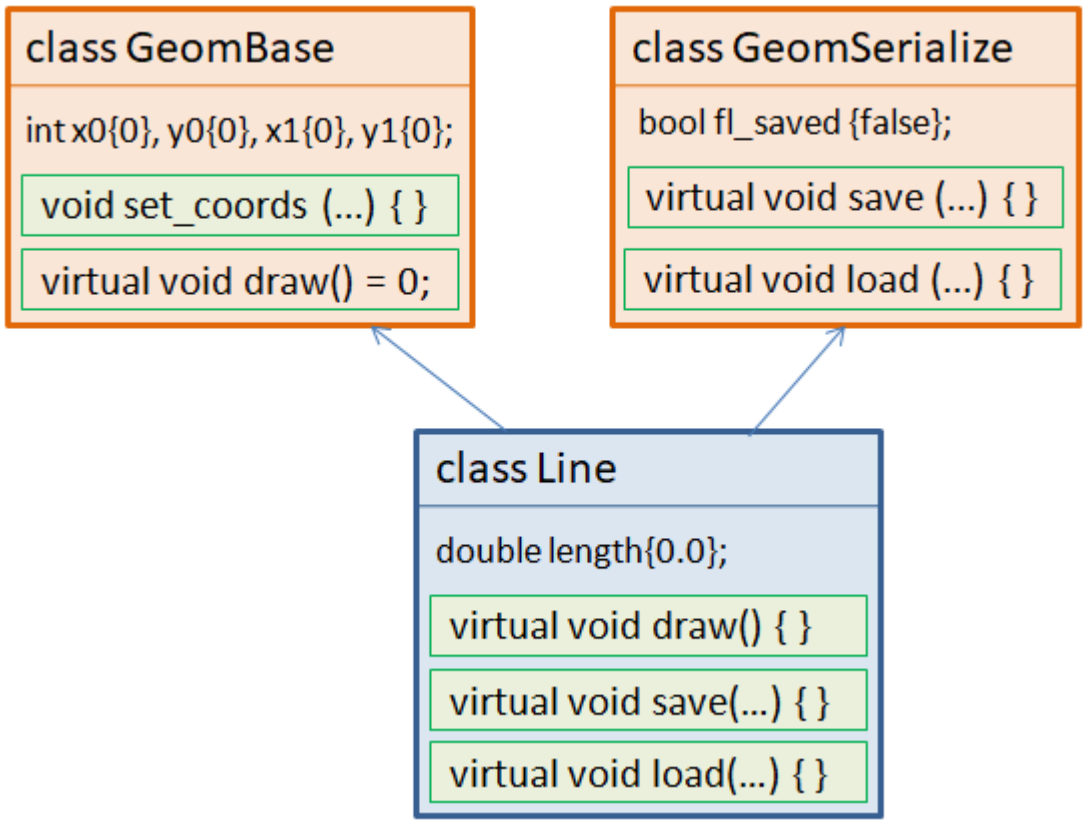
После этого в классе Line можно переопределить виртуальные методы save и load класса GeomSerialize. В самом простом варианте это можно сделать так:

```language-cpp
class Line : public GeomBase, public GeomSerialize {
...
public:
...
    virtual void save(std::ostream& os) const override
        { os.write((char *)this, sizeof(*this)); }

    virtual void load(std::istream& is) override
        { is.read((char *)this, sizeof(*this)); }
};
```

Если эти методы не переопределять в дочернем классе Line, то при компиляции возникла бы ошибка из-за отсутствия определения чисто виртуальных методов базового класса.

В результате мы получили следующую схему наследования классов:



Порядок вызовов конструкторов и деструкторов

Первый вопрос, который здесь возникает, в каком порядке вызываются конструкторы и деструкторы этих классов. Давайте посмотрим. Добавим в каждый класс публичный конструктор и деструктор с выводом сообщений:

```language-cpp
class GeomSerialize {
protected:
    bool fl_saved {false};
public:
    GeomSerialize()
        { std::cout << "GeomSerialize: constructor" << std::endl; }
    virtual ~GeomSerialize()
        { std::cout << "GeomSerialize: destructor" << std::endl; }
...
};

class GeomBase {
...
public:
    GeomBase(int a = 0, int b = 0, int c = 0, int d = 0)
        : x0(a), y0(b), x1(c), y1(d)
        { std::cout << "GeomBase: constructor" << std::endl; }
    virtual ~GeomBase()
        { std::cout << "GeomBase: destructor" << std::endl; }
...
};

class Line : public GeomBase, public GeomSerialize {
...
public:
    Line(int a = 0, int b = 0, int c = 0, int d = 0) : GeomBase(a, b, c, d)
        { std::cout << "Line: constructor" << std::endl; }
    ~Line()
        { std::cout << "Line: destructor" << std::endl; }
...
};
```

Если теперь создать в функции main объект класса Line:

```language-cpp
int main()
{
    Line ln(1, 2, 10, 20);
```

```
    return 0;
}
```

То в консоли увидим строчки:

```
GeomBase: constructor
GeomSerialize: constructor
Line: constructor
Line: destructor
GeomSerialize: destructor
GeomBase: destructor
```

То есть, сначала вызываются конструкторы базовых классов в порядке их указания при наследовании, затем, конструктор дочернего класса Line. А деструкторы отрабатывают в обратном порядке: сначала деструктор класса Line, затем, базовых классов GeomSerialize и GeomBase. Все вполне логично и ожидаемо.

Однако здесь есть один нюанс. **Конструктор дочернего класса Line вызывает конструктор базового класса GeomBase. При этом порядок вызовов конструкторов при делегировании следует записывать в том же порядке, что и классы при наследовании**. Например, если поменять порядок следования классов, то правильно было бы вызывать их конструкторы так:

```language-cpp
class Line : public GeomSerialize, public GeomBase {
...
public:
    Line(int a = 0, int b = 0, int c = 0, int d = 0) : GeomSerialize(),
GeomBase(a, b, c, d)
        { std::cout << "Line: constructor" << std::endl; }
...
};
```

Правда, современные компиляторы расположат вызовы конструкторов базовых классов в правильном порядке, вне зависимости от их записи при делегировании. Поэтому, можно специально не вызывать конструктор по умолчанию класса GeomSerialize:

```language-cpp
  Line(int a = 0, int b = 0, int c = 0, int d = 0) : GeomBase(a, b, c, d)
        { std::cout << "Line: constructor" << std::endl; }
```

Все по-прежнему будет работать. Но, по возможности, порядок все же следует соблюдать.

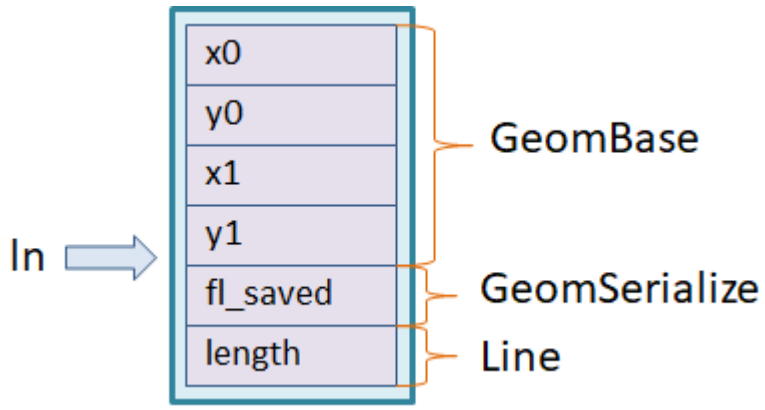
Объект дочернего класса при множественном наследовании

С порядком конструирования объектов дочерних классов при множественном наследовании мы разобрались. Следующий вопрос, что из себя представляет объект класса Line?

Пусть базовые классы при наследовании записаны в порядке:

```language-cpp
class Line : public GeomBase, public GeomSerialize { ... };
```

Тогда сначала создается объект класса GeomBase, а затем класса GeomSerialize. В результате объект дочернего класса будет содержать данные в следующем порядке:



Благодаря строгому порядку расположения переменных, мы можем преобразовать объект дочернего класса к указателю или ссылки любого базового. Например:

```language-cpp
int main()
{
    Line ln(1, 2, 10, 20);

    GeomBase* ptr_b = &ln;
    GeomSerialize* ptr_sz = &ln;

    GeomBase& lnk_b = ln;
    GeomSerialize& lnk_sz = ln;

    return 0;
}
```

При этом операции приведения типов в таких случаях прописывать не обязательно, компилятор языка C++ это сделает автоматически. Соответственно, через указатель или ссылку мы можем работать только с фрагментом соответствующего базового класса. Например:

```cpp
int main()
{
...
    ptr_b->draw();

    std::ofstream ofs("line.dat");
    lnk_sz.save(ofs);
    ofs.close();

    return 0;
}
```

На следующем занятии мы продолжим эту тему и рассмотрим некоторые проблемы и способы их решения при множественном наследовании.